Decaf: Decoupled Dataflows for In Situ High-Performance Workflows

Matthieu Dreher
Argonne National Laboratory
9700 S. Cass Ave.
Lemont IL 60439 USA
mdreher@anl.gov

Tom Peterka
Argonne National Laboratory
9700 S. Cass Ave.
Lemont IL 60439 USA
tpeterka@mcs.anl.gov

***Abstract*—Decaf is a dataflow system for the parallel communication of coupled tasks in an HPC workflow. The dataflow can perform arbitrary data transformations ranging from simply forwarding data to complex data redistribution. Decaf does this by allowing the user to allocate resources and execute custom code in the dataflow. All communication through the dataflow is efficient parallel message passing over MPI. The runtime for calling tasks is entirely message-driven; Decaf executes a task when all messages for the task have been received. Such a message-driven runtime allows cyclic task dependencies in the workflow graph, for example, to enact computational steering based on the result of downstream tasks. Decaf includes a simple Python API for describing the workflow graph. This allows Decaf to stand alone as a complete workflow system, but Decaf can also be used as the dataflow layer by one or more other workflow systems to form a heterogeneous task-based computing environment. In one experiment, we couple a molecular dynamics code with a visualization tool using the FlowVR and Damaris workflow systems and Decaf for the dataflow. In another experiment, we test the coupling of a cosmology code with Voronoi tessellation and density estimation codes using MPI for the simulation, the DIY programming model for the two analysis codes, and Decaf for the dataflow. Such workflows consisting of heterogeneous software infrastructures exist because components are developed separately with different programming models and runtimes, and this is the first time that such heterogeneous coupling of diverse components was demonstrated in situ on HPC systems.**

I. Introduction

Computational science involves a workflow of interconnected tasks, several of them being executed on a supercomputer, for example simulation, analysis, visualization, or user interaction. A workflow can be modeled as a directed graph, where the nodes of the graph are tasks, and the edges represent information exchanged between the tasks. We assume that the graph can have cycles; feedback loops can exist between tasks so that the result of a downstream task can be used to modify an upstream one (sometimes called computational steering).

We define *dataflow* to be the information exchange between tasks in a workflow. Traditionally, tasks exchange data through files. However, the growing mismatch between HPC computing rate and I/O bandwidth motivates shifting from file-oriented workflow models to *in situ workflows*, where dataflows are through memory or the supercomputer interconnect, avoiding the storage I/O bottleneck.

We designed our dataflow as a middleware layer separate from the workflow engine because a science campaign may consist of several workflow tools that need to cooperate, and this design allows one dataflow middleware to support multiple workflow tools simultaneously. However, several challenges need to be solved as a result of this design choice. The dataflow needs to manage the exchange between heterogeneous programming and data models because simulation and analysis tasks are often developed independently. Workflows can be instantiated with different numbers of resources (MPI ranks) per task, and the dataflow has to transform and redistribute data during the exchange between tasks. The dataflow runtime needs to support any generic directed graph because workflow graphs come in different shapes and sizes, ranging from a simple pipeline of two or three tasks to more complex topologies including fan-in, fan-out, and cycles.

Decaf is a new dataflow middleware for in situ workflows. Decaf is characterized by the following features, which constitute the contributions of this paper.

- A decoupled dataflow: hybrid of tight and loose coupling of tasks through an optional intermediate set of resources called a *link* that can be used to transform or redistribute data between producer and consumer.
- An efficient method for parallel data communication based on MPI using a simple C/C++ put/get API.
- A high-level Python API for defining the workflow graph so that Decaf can stand alone as a complete workflow system when it is not used in conjunction with other workflow tools.
- A design that allows composition of different workflow systems so that the workflow can be defined in a completely different software tool, with Decaf providing only the dataflow capability.
- A message-driven runtime for executing workflow tasks when all messages for a task have been received, which supports any directed graph topology, including cycles.
- Support for the MPMD (multiple program multiple data) capability of MPI to run multiple executables in the same job launch, a feature available on most supercomputing and cluster platforms.
- Demonstration in workflows consisting of several heterogeneous systems and simulation, analysis, and visualization application codes.

The reminder of this article is organized as follows. Section II reviews related work. Section III describes Decaf's design and key features. Section IV evaluates the capabilities and overhead of Decaf. Section V summarizes our accomplishments and briefly describes future work.

II. Background and Related Work

A. Types of In Situ Workflows

Our definition of in situ is broader than others' who limit it to tasks performed only in the same node or core of the computer. Some call this synchronous or tightly coupled as well [1]. We allow in situ to mean anywhere in the same HPC system during the same scheduled execution of a job. Hence, our use of the term includes what others have called in transit [2], [3] or coprocessing [4], [5], [6], in addition to the strictly synchronous turn-taking mode. Figure 1 illustrates the different in situ coupling modes.

We classify each pair of in situ coupled tasks (2 nodes connected by one link in the workflow graph) as being coupled by *time division* and *space division* [7]. In time division, the two tasks, producer and consumer, share the same resources and run sequentially in time. For example, a simulation (producer) and analysis (consumer) task may take turns operating on the same time step of data. The simulation computes the data for one time step and then waits while the analysis task processes the data, which then waits while the simulation generates the next step.

In space division, the producer and consumer run on different resources concurrently in time. Using the same simulation-analysis example, the simulation computes the first time step, copies or sends it to the consumer, and then computes the second time step while the analysis code processes the first time step. Both modes are used in practice, and there are time-space tradeoffs between them. Space division also must address flow control so that a fast producer does not overrun a slow consumer [8].

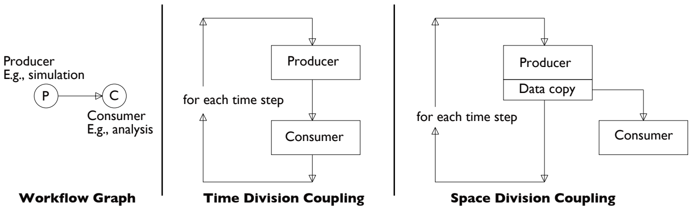


Fig. 1. Time-division and space-division in situ coupling of a producer and consumer task.

B. In Situ Workflow Runtimes

In situ middleware has been developed by various communities. Such middleware has to face performance, programability, and portability trade-offs highlighted in [9]. We characterize these tools according to their execution model of in situ tasks.

a) Time division: Scientific visualization is one of the main targets for in situ analytics, motivated by the need for high bandwidth and temporal resolution. Yu et al. [10] integrated a particle volume renderer synchronously in a combustion code (S3D). The renderer processing required 10% of the total execution time to produce images. Current production visualization tools include ParaView [11] and VisIt [12] for postprocessing. Both have an in situ library, respectively Catalyst [6] and Libsim [13], to process data from simulations. The goal of these libraries is to convert the internal data structures of the simulation to VTK data structures. The libraries then execute the rendering pipeline, usually synchronously, although other configurations are possible.

Other in situ middleware originates in the I/O community. Originally designed for I/O staging, these tools have subsequently coupled analysis and visualization applications together with simulations. Their API is intended to look to the simulation like a file write. ADIOS [14] is a common interface for multiple I/O methods. Although initially developed to store data efficiently in parallel, ADIOS now provides methods to share data between codes and transform data in situ along the I/O path [15].

b) Space division: Several frameworks using the ADIOS interface can transfer data asynchronously to dedicated resources. DataStager [2] can schedule data movement while the simulation is in its computational phase, avoiding unnecessary network contention. FlexPath [16] provides a publisher/subscriber model to exchange data between parallel codes having different numbers of processes ($M \times N$ communication pattern).

DataSpaces [17] implements a distributed shared-memory space with a publisher/subscriber interface for external applications. DataSpaces indexes data based on a space-filling curve. Data are then distributed among data servers based on their index. The index is used both for pushing data into DataSpaces and for retrieving data efficiently from it.

Damaris [18], [1] splits the global MPI communicator of a simulation to dedicated cores or nodes [19] in order to run the analysis concurrently with the application. The framework was used to stage I/O and for in situ visualizations. Functional partitioning [20] also uses dedicated cores for I/O with a FUSE interface.

HDF5 DSM [21] connects a simulation code to ParaView by using the HDF5 interface. The simulation and the ParaView servers can run in separate jobs. Both jobs read and write data to the shared virtual HDF5 file layer; steering is also possible.

c) Hybrid approaches: Both time division and space division have pros and cons, hence the need for flexibility in analytics placement [22]. Several solutions now support both time and space division. FlexIO [23] enables the user to perform analytics synchronously during the simulation or asynchronously on dedicated cores or nodes. Bennett et al. [3] combined computations in time division followed by other processing on dedicated nodes using DART servers [24] to transfer data. They applied several analysis algorithms to the combustion code S3D this way.

FlowVR [8] couples parallel codes to form a graph of tasks. The user manually sets the placement of each task allowing any placement strategy. Each task is usually a separate executable. Communication between parallel codes is managed by a daemon running on each node. Decaf also adopts a hybrid approach. Similar to FlowVR, Decaf composes multiple executables to form a workflow. However its execution model is designed for current supercomputer environments, and it does not rely on a sepate daemon to manage the graph execution.

C. *Workflow Graph Definition*

ADIOS [14] describes the data model and the transport method in a separate XML file. Swift [25] is a script language to write a parallel program; the runtime then analyzes the program to extract parallel tasks and executes the workflow graph. FlowVR [8] has a simple Python interface to describe how to launch individual tasks and how to link them to form a graph. The Python script then generates intermediate XML files read by the runtime to launch each task and generate the appropriate communication channel. Decaf has a similar Python interface as FlowVR to describe the graph. However, the description is simplified: the user describes nodes and links as serial entities while Decaf takes care of their parallelism. Other workflow engines are tailored for distributed computing, such as PyCOMPSs [26] and Pegasus [27]. PyCOMPSs uses annotations in a Python program to generate tasks and schedule them within jobs. Pegasus include scripts to generate common workflow patterns from templates.

III. Design

A. *Decaf Dataflow*

The main building block of a Decaf workflow is a *dataflow*. A dataflow is the association of a producer, consumer, and a communication object to exchange data between the producer and consumer. Producers and consumers, called nodes in Decaf, are parallel user-supplied codes such as simulation and analytics. For instance, Gromacs [28], a molecular dynamics simulation, typically generates atom positions, and scientists use tools such as VMD [29] to visualize and analyze the atom positions. In an in situ context, Gromacs and VMD communicate directly through memory. Data structures between the two codes are often different, and some data adaptation might be required. This can be as simple as converting data units (from nanometers to angstroms) to more complex data rearrangements such as mapping a simulation data structure to a VMD data structure.

These data transformations can require extensive computations. To address these needs, a Decaf dataflow includes an intermediate staging area with computational resources. We call the staging area a *link*. A link is an intermediate parallel program transforming data between a parallel producer and a parallel consumer.

Figure 2 shows the organization of a dataflow in Decaf. More advanced workflow graphs can be obtained by combining multiple dataflows. The dataflow is a simple graph with two nodes and a link. All three are MPI programs, each with its own MPI communicator. The dataflow creates two additional communicators: producer-link and link-consumer. Section III-C describes how data are exchanged through the communicators. Section III-D explains how to describe such graphs and how the runtime executes them.

The link resources are optional; one can disable the link when no data manipulations are required between the producer and consumer. In that case, the user does not assign any MPI ranks to the link, and the runtime creates only one communicator directly between the producer and the consumer.

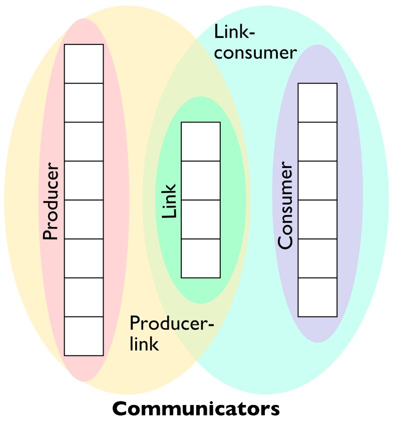


Fig. 2. Decaf forms 5 communicators for a dataflow: one communicator each for producer, consumer, and link, plus two more communicators for the overlap between producer-link and link-consumer.

```
int main(int argc,
         char** argv)
{
    // define the workflow
    Workflow workflow;
    make_wflow(workflow);

    // initialize MPI and Decaf
    MPI_Init(NULL, NULL);
    Decaf* decaf = new Decaf(MPI_COMM_WORLD, workflow);

    // run the task
    vector<pConstructData> in_data;
    while (decaf->get(in_data))
    {
        // process the input data

        // send the results outbound
        pConstructData out_data;
        decaf->put(out_data);
    }

    // end the task
    decaf->terminate();

    // cleanup
    delete decaf;
    MPI_Finalize();
    return 0;
}
```

Listing 1. A typical node program constructs the Decaf object and executes a node task. The task waits for all its inputs to be satisfied and then processes the received data. It can safely do this in an infinite loop because the Decaf `get()` function returns false upon termination

B. *Code Modification*

Decaf differentiates between node (producer, consumer) and link code. Decaf nodes are existing codes such as simulation or analytics. In order to minimize the code modifications necessary to integrate these codes. Decaf has a simple put/get model that allows tasks to send/receive data to/from the rest of the workflow. Listing 1 gives the general structure of a node program.

After the initialization of the runtime (Section III-D), the user starts a loop over incoming data. The `get(in_data)` call waits until data are received on all inbound links or a termination message is received. In the latter case, the code exits the iteration loop. The user then processes the incoming data and possibly sends new data by calling `put(out_data)`. `pConstructData` is the handler of the data model described in Section III-C. The `terminate()` call signals to the runtime to exit the application. If the node is a source (no inbound link), the user simply does not call `get(in_data)`. If the node is a sink (no outbound link), the user does not need to call

```
int main(int argc,
         char** argv)
{
    // define the workflow
    Workflow workflow;
    make_wflow(workflow);

    // initialize MPI and Decaf
    MPI_Init(NULL, NULL);
    Decaf* decaf = new Decaf(MPI_COMM_WORLD, workflow);

    // cleanup
    delete decaf;
    MPI_Finalize();
    return 0;
}

void link_callback(void* args,
                   Dataflow* dataflow,
                   pConstructData in_data)

    // process the input data
    ...

    // send the results outbound
    pConstructData out_data;
    decaf->put(out_data);

}
```

Listing 2. A typical link program constructs the Decaf object and provides a link callback function.

put(out_data). This API is available in C and C++.

Links differ from nodes because they are not meant to run a loop but rather to manage the flow of data. The user provides to the runtime a callback function for each link in the workflow graph. Recall that a link has associated computational resources. Every time a node sends data, the link executes the callback function provided by the user.

Listing 2 describes a simple link skeleton. The link execution is managed by Decaf, which calls link_callback when necessary. The callback accesses the data in the dataflow, processes them, and forwards them to the consumer of the dataflow. Some in situ infrastructures such as PreData [30] also allow processing data on the fly by using *codelets*. Decaf differs in that the links are full C/C++ code with dedicated computational resources.

C. Data Model and Redistribution Components

Decaf relies on Bredala [31] to describe data and transfer data between MPI programs. Bredala protects the semantic integrity of a data model during split and merge operations using the notion of a semantic item. This is the smallest subset of data that contains all the fields of the original data structure and still preserves its semantics. Bredala provides a safe way to access, extract, and merge semantic items within a data model. Bredala comes with four common redistribution strategies: round-robin, contiguous, spatial, and block redistribution (Figure 3).

Decaf uses both the data model and the redistribution components of Bredala. The data model serves as the data interface to exchange data between tasks within the workflow. For each dataflow, Decaf creates two redistribution components: one between the producer and the link and one between the link and the consumer. When calling put(), Decaf passes the data to the corresponding redistribution component that manipulates and transmits the data to its destination. When calling get(), Decaf receives data on the corresponding redistribution component and transmits it to the task.

```
w = nx.DiGraph()

# Task declaration
w.add_node('node0', start_proc=0,  nprocs=4)
w.add_node('node1', start_proc=7,  nprocs=2)
w.add_node('node2', start_proc=11, nprocs=1)

# Dataflow declaration
w.add_edge('node0', 'node1', start_proc=4, nprocs=3,
            func='dflow', path=mod_path,
            prod_dflow_redist='count',
            dflow_con_redist='count')
w.add_edge('node1', 'node2', start_proc=9, nprocs=2,
            func='dflow', path=mod_path,
            prod_dflow_redist='count',
            dflow_con_redist='count')

wf.workflowToJson(w, mod_path, "linear3.json")
```

Listing 3. Code sample to generate a 3-node pipeline.

D. Graph Description and Runtime Execution

A Decaf workflow is the composition of multiple dataflows. For instance, to create a pipeline of three tasks, the user declares two dataflows, where the second task is both consumer in the first dataflow and producer in the second dataflow. Decaf does not apply any constraints on the graph topology. In particular, the user can define cycles in the graph for steering scenarios. Moreover, a particular task can be a producer and/or consumer in multiple dataflows.

The user describes the workflow graph in a Python script. Listing 3 presents the code generating a three-task pipeline. First, the user describes individual tasks with a name and a set of resources (MPI ranks). Second, the user describes the dataflow linking two tasks (producer, consumer), a set of resources (MPI ranks for the link), and the choice of redistribution strategy. The same task can be used by several dataflows as producer or consumer. Once all the tasks are described, the user calls workflowToJson to generate a JSON file. That file contains an intermediate representation of the workflow read by the Decaf runtime upon initialization. The graph description is done prior to the launch of the workflow.

In a Decaf workflow, every task is an MPI program. Each task can be a separate program, or all tasks may be combined in a single program. We rely on the MPMD (multiple program multiple data) capability of MPI to launch all the programs. With this method, MPI_COMM_WORLD is shared among all the executables. Based on the information provided by the JSON file, Decaf creates five communicators (Figure 2) per dataflow: one for the producer, link, and consumer, and one for each redistribution component. This method requires existing codes such as simulations only to replace their equivalent of MPI_COMM_WORLD by the communicator provided by Decaf.

The link functions are provided by the user and called by Decaf. The user provides a library path and function name to call when declaring a link so the runtime can load the user's functions. The link code can be executed in separate or the same MPI ranks as the producer or consumer.

IV. Evaluation

We evaluated Decaf in two workflows: one from molecular dynamics and one from cosmology. With the molecular dynamics example, we evaluate the steering capabilities of Decaf and its interoperability with other in situ middleware to create complex workflows. With the cosmological example, we test

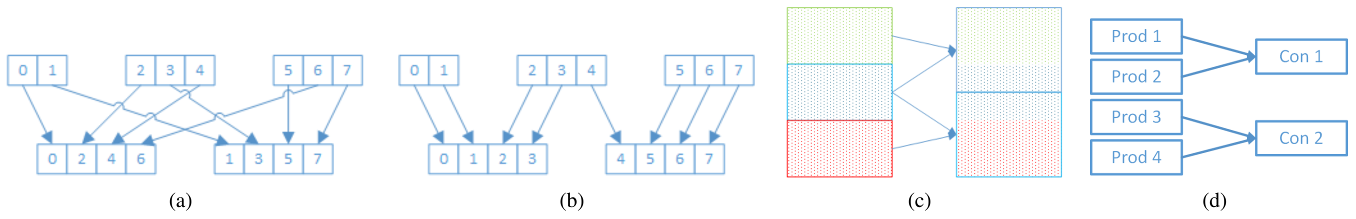


Fig. 3. Redistribution components available in Bredala. (a) The round-robin component distributes the data item by item in a cyclic manner. (b) The contiguous component redistributes all the items while preserving their order. (c) The spatial component divides a global domain into subdomains and attributes each item to a particular subdomain. (d) The block component takes the data model as it is and sends it to a destination based on the MPI ranks.

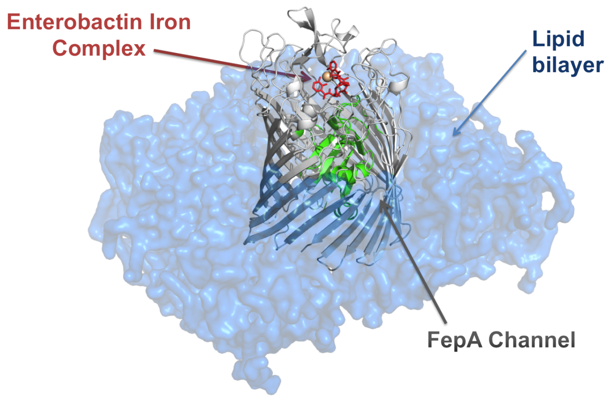


Fig. 4. The FepA protein (grey), a channel on the membrane (blue) of particular cells. Several compounds, such as an iron complex (red), can traverse the membrane of the cell thanks to this channel. We study the traversal of this complex and the behavior of the secondary structures (green) during the traversal.

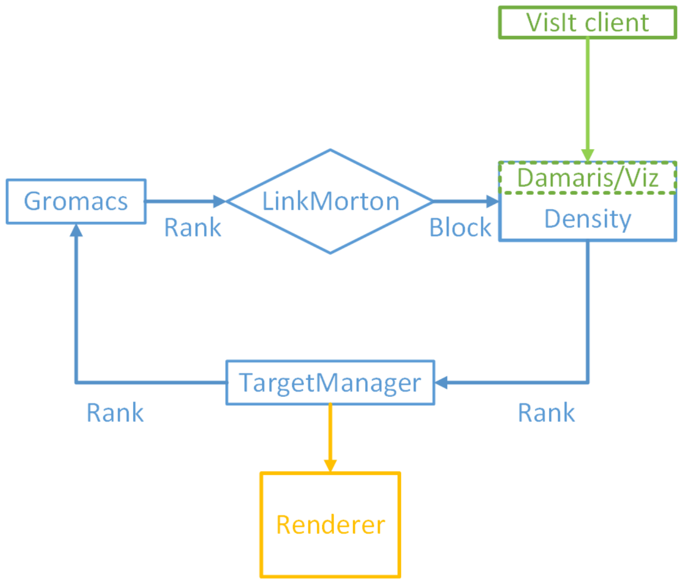


Fig. 5. Steering workflow with Decaf (blue), Damariz/Viz (green), and FlowVR (orange). The steering part is managed by Decaf while visualizations are handled by Damaris and FlowVR. Decaf and FlowVR tasks and communications are in space-division mode (blue and orange arrows) while Damaris execution is executed in time-division mode during the execution of `Density`.

the scalability of our framework and its capacity to handle large, complex data structures.

A. *Molecular Dynamics*

Our first example is steering a molecular dynamics simulation to trigger a biological mechanism. We study the FepA protein (Figure 4), a channel in the periplasm of Gram-negative bacteria. Compounds can traverse the membrane of the cell through this channel. In this case, we study the traversal of an iron complex. This mechanism is of interest to biologists because drugs pass through these channels.

Steering the simulations allows a biologist to push a complex within a channel and accelerate the traversal process. The scientist applies external forces to a subset of atoms to guide the simulation toward a desired state. Previous works based on this method demonstrated the interactive traversal of the iron complex through the FepA channel [32] and with batch simulations [33].

Workflows supporting computational steering pose several challenges. First, a steering workflow needs to support cycles in the workflow graph, which are difficult to handle because they potentially can generate deadlocks. Second, the steering process must be performed asynchronously to avoid blocking the simulation. Third, because multiple types of interactions may be needed (force feedback, visualization, density computation, etc), the middleware must be able to integrate different tools coming from different communities into a single workflow.

In this experiment, we implement a steering workflow with Decaf to guide the iron complex toward the FepA channel, based on the implementation proposed in [33]. Figure 5 summarizes the workflow. First, we integrate Decaf API calls into Gromacs [28], a molecular dynamics simulation code, to add external forces and expose atom positions to the rest of the workflow. Second, in a link (`LinkMorton`), we compute a Morton code [34] for each atom particle. Third, we compute a 3D density grid (`Density`). Fourth, we compute the force (`TargetManager`) to guide the system. The trajectory for the complex to follow is defined by a list of target positions, provided by the user, forming a path. We used a path-finding algorithm [35] based on the density grid previously computed to guide the iron complex from one target to the next. In this workflow, we use only one link to compute the Morton indices. We configure the rest of the dataflows to skip the link because no further data manipulations are necessary.

At each step of the workflow, we modify the fields of the data model. First, we send atom positions and their indices from the simulation to the link with a block redistribution strategy. Second, we add the Morton indices and use a block redistribution strategy to reorganize the data to compute the density grid. Third, we remove the Morton indices from the data model (no longer necessary) and append a 3D grid to the data model. We send the result to `TargetManager`, which broadcasts forces to the simulation with a block redistribution strategy.

In steering processes, visualization is an important tool for debugging and following the simulation's progress. For this application, we rely on two existing tools to perform in situ visualization. First, we connected our workflow to the renderer used in [33] based on Hyperballs [36] to visualize the molecular model and the state of the steering system (Figure 6(a)). This visualization is the most convenient for the biologist to guide the complex because the user can navigate within the molecular structure and track the iron complex. We modified the `TargetManager` module to extract atom positions and send them to a FlowVR context for visualization. This visualization is performed in the space-division mode. We use FlowVR because it provides the components to decouple the processing rate of the two components and because the biologists use FlowVR for the renderer. Second, we used Damaris/Viz [1] in the time-division mode to visualize the density grid with VisIt (Figure 6(b)). The density grid is extracted in parallel in the `Density` task. The visualization of the density grid shows the biologist where low density areas are located, which are good candidates for the iron complex to go through. Because no tools support both visualizations, we need to combine several visualization packages in the same workflow.

The following tests were conducted on Froggy, a 138-node cluster from the Ciment infrastructure. Each compute node is equipped with 2 eight-core processors, Sandy Bridge-EP E5-2670 at 2.6 GHz with 64 GB of memory. Nodes are interconnected through an FDR InfiniBand network. FlowVR 2.1 and Gromacs 4.6 are compiled with Intel MPI 4.1.0. For all experiments we used the molecular model of the FepA composed of about 70,000 atoms. For all scenarios, we extracted atom positions from the simulation every 100 iterations.

We first evaluate the impact of modifying Gromacs to run within Decaf. Our goal is to evaluate the overhead introduced by using Decaf. We created a workflow where Gromacs is the only task. We modified Gromacs to receive forces and extract atom positions and their respective indices. We also replaced all instances of `MPI_COMM_WORLD` by the communicator provided by Decaf, although for the first experiment, the communicator provided by Decaf is equivalent to `MPI_COMM_WORLD`. The overall code modifications constitutes roughly 150 additional lines of code.

Figure 7 presents the timing decomposition of Gromacs with increasing numbers of cores (strong scaling). The timings represent a full I/O cycle, extracting atom positions every 100 iterations. `Gromacs` is the average time of 100 simulation iterations without any Decaf calls. `Get` and `Put` are the average time of a call to Decaf `get()` and `put()`, respectively. When a task has no input or output communication channel, Decaf calls return immediately. In all cases, the time spent in Decaf operations is less than 0.001%. This experiment shows that the same version of the code can be used as a production standalone executable or as part of a workflow without any perceivable performance impact. The Decaf framework adds no signifigant overhead.

Our second set of experiments studies the impact of our steering pipeline on Gromacs performance. We created the workflow as described in Figure 5 but without any visualization. Our goal is to evaluate the capability of Decaf to efficiently perform computations in space-division mode without significant overhead on the simulation performance.

We ran each task using a space-division strategy. We allocated one node as a staging node and the remaining nodes as simulation nodes. We hosted the `Density` (four cores) and `TargetManager` (one core) tasks on the staging node. We limited the `Density` task to four cores because the density grid is relatively small (170x80x180). Gromacs runs on each simulation node using 15 cores per node out of 16. The `LinkMorton` task runs on the remaining core on each simulation node to preserve data locality. We configured MPI to bind each MPI rank to a given core to avoid process migration and reduce performance variability.

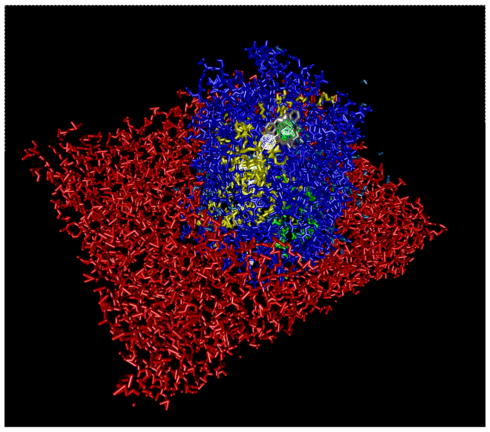
Figure 8 presents the timing of the Gromacs execution and the steering process. As before, we give the timing of a full I/O cycle (100 simulation iterations). `Steering` indicates the accumulated execution time of the `LinkMorton, Density`, and `TargetManager` tasks.

The steering processes has a time budget of 100 simulation iterations to complete before blocking the simulation. In our setup, the steering pipeline requires between 30 and 60 ms. At 480 cores, the simulation requires on average 277 ms to compute 100 iterations. In all cases, the time spent by Gromacs in Decaf operations represents less than 0.1% of a full I/O cycle. This experiment shows that Decaf is able to efficiently overlap in situ computations in the space-division mode with negligible impact on the simulation performance.

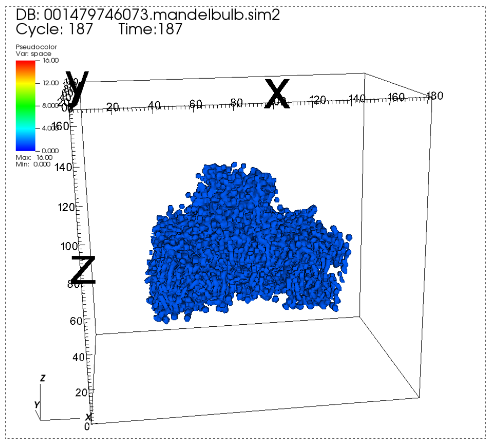
We also evaluated the cost of in situ visualization. Our first rendering method uses the FlowVR molecular renderer. FlowVR allows us to send atom positions directly to the visualization without blocking the pipeline. We used the same setup as for the previous experiment with 480 cores for the simulation. As before, the time spent by Gromacs in `get()` and `put()` calls represents less than 0.1% of a full I/O cycle. The steering process, including the transfer to the FlowVR visualization, requires in the worst case 63 ms, while the simulation takes 277 ms to complete a cycle. With this setup, the visualization computation time is completely overlapped by the simulation computations.

Our second rendering method used Damaris/Viz and libsim VisIt to display the density grid computed in the `Density` task. We used the time-division mode: the rendering time was directly added to the steering time. With this scenario, we visualized the density grid using isosurface rendering. The rendering was performed only if a VisIt client is connected to the VisIt server.

Figure 9 presents the time decomposition of Gromacs over numerous iterations. *Gromacs* is the execution of Gromacs including Decaf operations. *Get* and *put* give the respective timings of `get()` and `put()`. *Visit* gives the time spent in VisIt operations. We notice three distinct phases. The first phase, between iteration 0 and 19,400, shows Gromacs with an average execution time of 240 ms and almost no time spent



(a)



(b)

Fig. 6. (a) Visualization of the molecular system with FlowVR. (b) Visualization of the density grid with Damaris/Viz.

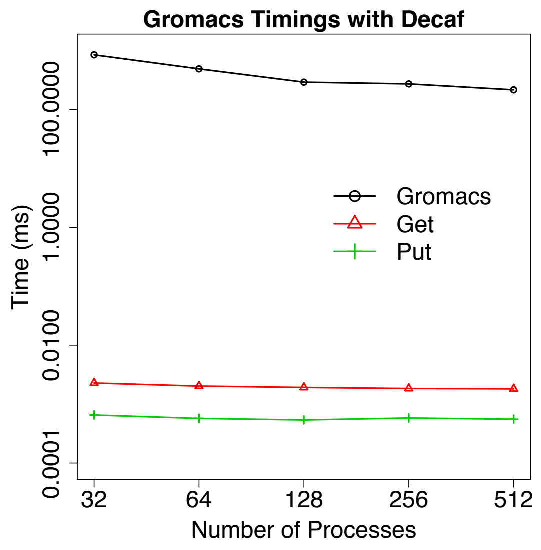


Fig. 7. Time decomposition of Gromacs modified with Decaf.

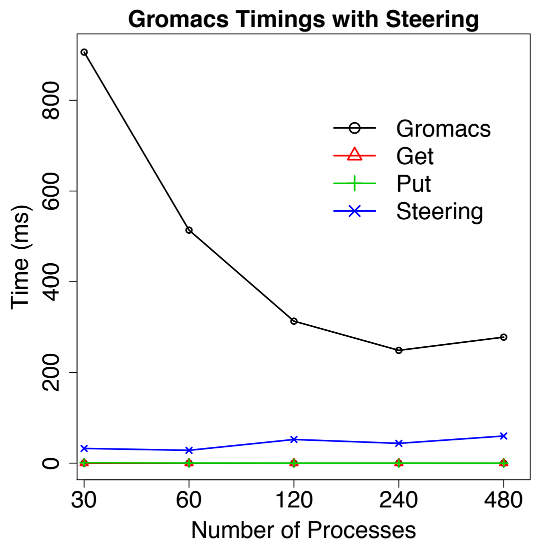


Fig. 8. Time decomposition of Gromacs performance modified with Decaf and connected to the steering pipeline without visualization. The measured time is the average over a full I/O cycle (100 iterations).

in `get()` or `put()`. VisIt requires only 4 ms in this phase because no clients are connected to the servers. In the second phase, between iteration 19,400 and 27,200, we connect a VisIt client to the workflow. Consequently, Gromacs execution time increases significantly. Most of the execution time is spent in `get()`, which blocks until the full steering process is completed including the VisIt rendering. The isosurface rendering method is costly, between 1,000 ms and 3,933 ms, and increases the steering time because the rendering is performed synchronously in time-division mode. As a reminder, the time budget for the simulation is 277 ms on average. Any computation longer than this threshold, which is the case with VisIt, blocks the simulation. The third phase, from iteration 27,200 to the end, presents results similar to the first phase. We disconnected the VisIt client; VisIt rendering computations stop, and its execution time is reduced to 4 ms as during the first phase.

This experiment demonstrates that Decaf is able to create complex workflows, integrate the user in the loop, and perform asynchronous computation at a low cost. It also shows that Decaf is able to interact with other in situ middleware such as FlowVR and Damaris/Viz which are more appropriate tools for specific visualization tasks. The code to run these experiments is available in open source.[1]

B. Cosmology

The second use case is the analysis of dark matter tracer particles from a Vlasov-Poisson N-body cosmology code. In this workflow, we focused on the conversion of particle data to the deposition of particle density onto a regular 2D and 3D grid, using a Voronoi tessellation as an intermediate step. For high dynamic range data such as dark matter particles, computing the Voronoi tessellation first produces more accurate density estimates than less expensive methods that compute the density directly from the particle data [37]. The data model produced by the cosmology code, a set of particles, is transformed into

[1]https://bitbucket.org/matthieu_dreher/gromacs_interactive

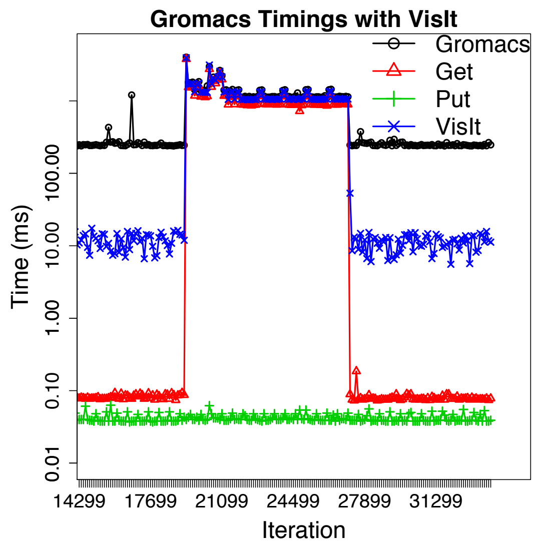


Fig. 9. Time decomposition of Gromacs as a function of iteration count. *Gromacs* is the execution of Gromacs including Decaf operations. *Get* and *put* give the respective timings of `get()` and `put()`. *VisIt* gives the time spent in VisIt operations.

the intermediate data model of the Voronoi tessellation, an unstructured polyhedral mesh, before being converted into a 2D or 3D regular grid of density scalar values.

The N-body cosmology code is HACC [38]. The tessellation and density estimator codes are built on the Tess library [39], which in turn is built on the DIY data-parallel programming model [40]. The three tasks are coupled with a link between each task in a 3-node linear workflow graph using Decaf. The link between the simulation and tessellation nodes rearranges the particles from the structure of arrays (SOA) format produced by HACC to an array of structures (AOS) format required by Tess. The link between the tessellation and density estimator simply forwards the polyhedral mesh data model without modifying it.

All three tasks are parallel MPI programs that scale to large numbers of MPI processes. The intermediate tessellation has a large memory footprint, approximately 15 times as large as the simulation data, making it necessary to compute the tessellation on a separate set of compute nodes from the simulation or density estimation. The density image (Figure 10) is a fraction of the simulation data size; but because the density estimator ingests the Voronoi tessellation, it begins with the same memory footprint as the tessellation. Because of the large memory footprints of the analysis tasks and the desire to perform analysis simultaneously with simulation, we configured all the nodes and links with space division to use separate resources. In the following experiment, we selected equal numbers of processes for all tasks and links.

The tests were conducted on the IBM Blue Gene/Q Vesta machine at the Argonne Leadership Computing Facility at Argonne National Laboratory. Vesta is a testing and development platform consisting of 2K nodes, each node with 16 cores (PowerPC A2 1.6 GHz), 16 GB RAM, and 64 hardware threads. We ran 8 MPI processes per compute node. The Clang compiler, based on LLVM version 3.9, was used to compile the code with -O3 optimization.

Figure 11 show the performance of a strong-scaling test in log-log scale. In our test, we used 128^3 particles estimated onto a 512^2 2D output grid. The vertical axis is the time to simulate a total of 100 time steps and to compute the tessellation and density estimate every 10 time steps. The horizontal axis is the total number of MPI processes for the entire workflow. The curves in the plot include total (end-to-end) time as well as the time for each of the three tasks.

The overall strong-scaling efficiency is approximately 50% from 40 to 320 processes. Past that point, the diminishing number of particles per process combined with the increasing imbalance between processes reduces the scalability of the tessellation, which in turn affects the rest of the workflow. For example, at 1,280 processes, the final time step of the simulation produces only 16 particles in the least-loaded process and 1,972 particles in the most-loaded one. The load imbalance as dark matter particles cluster into halos is expected; using a k-d tree instead of a regular grid of blocks to perform the tessellation [41] will improve this situation. The overall small number of particles per process implies that we have exceeded the required resources for this size problem; we intend to simulate larger problems with more initial particles in the future.

Several trends are evident in the timing measurements. The times for the individual tasks are similar to each other. The reason is that the data dependencies between tasks and their intertask communication synchronizes them. Moreover, the tasks overlap in time so that the total time is only slightly longer than that of the longest task, which is density estimation, but much less than the sum of the parts. This is especially true over the total of multiple time steps and effectively hides the analysis time from the simulation. One curious data point is at 640 processes, where the time increases from 320 processes; we are investigating the cause of this slowdown .

V. Conclusion

We introduced Decaf, a lightweight middleware for coupling in situ tasks to form workflows. Decaf makes it easy to modify existing codes such as simulation or analytics codes with a simple API. It is also simple to describe the workflow in Python as a directed graph where nodes are tasks and edges are communication channels. The Decaf runtime takes care of building the communication channels and scheduling the data exchanges. Decaf does not impose any constraints on the graph topology and can manage graphs with cycles.

We evalutated Decaf's capabilities and interoperability with various examples. The first was a steering scenario merging a molecular dynamics simulation, user analytic codes, and visualizations with FlowVR and Damaris/Viz. With our second example, we studied Decaf's scalability up to 1,280 cores with a cosmology example involving complex data structures to exchange.

Parallel producer and consumer tasks in a workflow often run at difference paces. For instance, a consumer might not be able to process incoming data fast enough. In our future work, we will incorporate buffering capabilities between producers and consumers to manage disparate data rates. We also plan to add data consistency checks when creating a workflow graph. The goal is to ensure that a consumer requiring a particular data

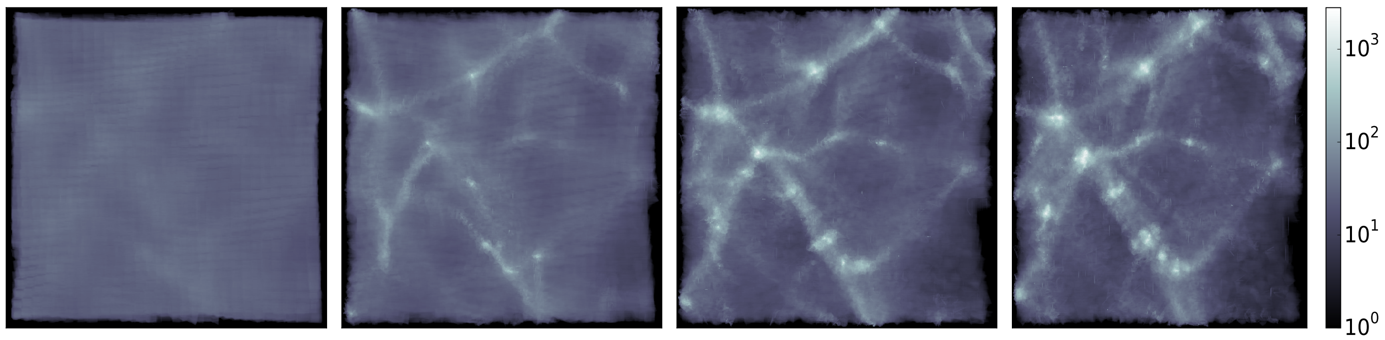


Fig. 10. (Left to right) Output of cosmology analysis workflow at time steps 10, 30, 60, and 100.

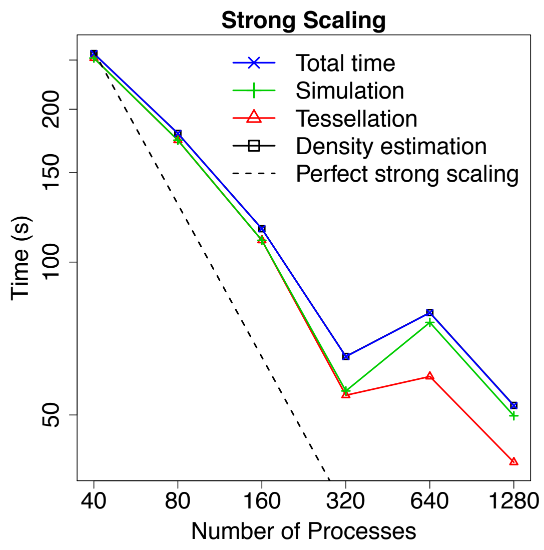


Fig. 11. Strong scaling of the cosmology simulation-tessellation-density estimation workflow shows good scaling efficiency until the small number of particles per process and load imbalance of the tessellation reduce efficiency. The time of the analysis tasks is effectively overlapped with the simulation time.

model is connected to a producer providing a corresponding data model.

The Decaf project is available in open source.[2]

ACKNOWLEDGMENTS

This work is supported by Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357, program manager Lucy Nowell. We are indebted to Katrin Heitmann and Salman Habib for use of the HACC simulation and feedback on the cosmology evaluation.

Some of the computations presented in this paper were performed using the Froggy platform of the CIMENT infrastructure (https://ciment.ujf-grenoble.fr), supported by the Rhône-Alpes region (GRANT CPER07_13 CIRA) and the Equip@Meso project (reference ANR-10-EQPX-29-01) of the programme Investissements d'Avenir supervised by the ANR.

[2]https://bitbucket.org/tpeterka1/decaf

REFERENCES

[1] M. Dorier, R. Sisneros, Roberto, T. Peterka, G. Antoniu, and B. Semeraro, Dave, "Damaris/Viz: a Nonintrusive, Adaptable and User-Friendly In Situ Visualization Framework," in *LDAV - IEEE Symposium on Large-Scale Data Analysis and Visualization*, Atlanta, United States, Oct. 2013. [Online]. Available: https://hal.inria.fr/hal-00859603

[2] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "Datastager: Scalable Data Staging Services for Petascale Applications," in *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '09. New York, NY, USA: ACM, 2009, pp. 39–48. [Online]. Available: http://doi.acm.org/10.1145/1551609.1551618

[3] J. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–9.

[4] *GLEAN*. CRC Press, Taylor and Francis Group, November 2014.

[5] R. Michel, S. Cameron, C. Kedar, S. E. Seegyoung, A. M. Benjamin, M. Jeffrey L., S. Onkar, L. Raymond M., S. Mark S., and J. Kenneth E., "Scalable implicit flow solver for realistic wing simulations with flow control," *Computing in Science & Engineering*, vol. 16, no. 6, pp. 13–21, 2014.

[6] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. Jansen, "The Paraview Coprocessing Library: a Scalable, General Purpose In Situ Visualization Library," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, Oct 2011, pp. 89–96.

[7] "The in situ terminology project," Feb 2016, https://ix.cs.uoregon.edu/h̃ank/insituterminology/index.cgi?n=Phase1B.Phase1BProposedInSituCategorizations.

[8] M. Dreher and B. Raffin, "A Flexible Framework for Asynchronous In Situ and In Transit Analytics for Scientific Simulations," in *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Chicago, United States, May 2014. [Online]. Available: https://hal.inria.fr/hal-00941413

[9] M. Dorier, M. Dreher, T. Peterka, J. M. Wozniak, G. Antoniu, and B. Raffin, "Lessons learned from building in situ coupling frameworks," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. ACM, 2015, pp. 19–24.

[10] H. Yu, C. Wang, R. Grout, J. Chen, and K.-L. Ma, "In situ visualization for large-scale combustion simulations," *Computer Graphics and Applications, IEEE*, vol. 30, no. 3, pp. 45–57, 2010.

[11] J. Ahrens, B. Geveci, and C. Law, "36 paraview: An end-user tool for large-data visualization," *The Visualization Handbook*, p. 717, 2005.

[12] S. Ahern, E. Brugger, B. Whitlock, J. S. Meredith, K. Biagas, M. C. Miller, and H. Childs, "Visit: Experiences with sustainable software," *arXiv preprint arXiv:1309.1796*, 2013.

[13] B. Whitlock, J. M. Favre, and J. S. Meredith, "Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System," in

Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization, ser. EGPGV '11. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2011, pp. 101–109. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV11/101-109

[14] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu, "Hello ADIOS: the Challenges and Lessons of Developing Leadership Class I/O Frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3125

[15] D. Boyuka, S. Lakshminarasimham, X. Zou, Z. Gong, J. Jenkins, E. Schendel, N. Podhorszki, Q. Liu, S. Klasky, and N. Samatova, "Transparent I Situ Data Transformations in ADIOS," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, May 2014, pp. 256–266.

[16] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-based publish/subscribe system for large-scale science analytics," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, May 2014, pp. 246–255.

[17] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: an Interaction and Coordination Framework for Coupled Simulation Workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10)*. New York, NY, USA: ACM, 2010, pp. 25–36.

[18] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O," in *CLUSTER - IEEE International Conference on Cluster Computing*. IEEE, Sep. 2012.

[19] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, and L. Orf, "Damaris: Addressing Performance Variability in Data Management for Post-Petascale Simulations," *ACM Transactions on Parallel Computing (ToPC)*, 2016.

[20] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman, "Functional partitioning to optimize end-to-end performance on many-core architectures," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.28

[21] J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, and J.-G. Piccinali, "Parallel Computational Steering and Analysis for HPC Applications using a ParaView Interface and the HDF5 DSM Virtual File Driver," in *Eurographics Symposium on Parallel Graphics and Visualization*, T. Kuhlen, R. Pajarola, and K. Zhou, Eds. The Eurographics Association, 2011.

[22] F. Zheng, H. Abbasi, J. Cao, J. Dayal, K. Schwan, M. Wolf, S. Klasky, and N. Podhorszki, "In-situ i/o Processing: A Case for Location Flexibility," in *Proceedings of the Sixth Workshop on Parallel Data Storage*, ser. PDSW '11. New York, NY, USA: ACM, 2011, pp. 37–42. [Online]. Available: http://doi.acm.org/10.1145/2159352.2159362

[23] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu, "FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 320–331.

[24] C. Docan, M. Parashar, and S. Klasky, "Enabling high-speed asynchronous data extraction and transfer using dart," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 9, pp. 1181–1204, 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.1567

[25] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A Language for Distributed Parallel Scripting," *Parallel Computing*, vol. 37, no. 9, 2011.

[26] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "Pycompss: Parallel computational workflows in python," *International Journal of High Performance Computing Applications*, 2015. [Online]. Available: http://hpc.sagepub.com/content/early/2015/08/19/1094342015594678.abstract

[27] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus: a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015, funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575. [Online]. Available: http://pegasus.isi.edu/publications/2014/2014-fgcs-deelman.pdf

[28] S. Pronk, S. Pall, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel, B. Hess, and E. Lindahl, "Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit," *Bioinformatics*, vol. 29, no. 7, pp. 845–854, 2013.

[29] W. Humphrey, A. Dalke, and K. Schulten, "VMD – Visual Molecular Dynamics," *Journal of Molecular Graphics*, vol. 14, pp. 33–38, 1996.

[30] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "PreDatA - Preparatory Data Analytics on Peta-Scale Machines," in *Parallel Distributed Processing (IPDPS'10)*, 2010, pp. 1–12.

[31] M. Dreher and T. Peterka, "Bredala: Semantic Data Redistribution for In Situ Applications," in *CLUSTER - IEEE International Conference on Cluster Computing*. IEEE, Sep. 2016.

[32] M. Dreher, M. Piuzzi, T. Ahmed, C. Matthieu, M. Baaden, N. Férey, S. Limet, B. Raffin, and S. Robert, "Interactive Molecular Dynamics: Scaling up to Large Systems," in *International Conference on Computational Science, ICCS 2013*. Barcelone, Spain: Elsevier, Jun. 2013. [Online]. Available: https://hal.inria.fr/hal-00809024

[33] M. Dreher, J. Prevoteau-Jonquet, M. Trellet, M. Piuzzi, M. Baaden, B. Raffin, N. Férey, S. Robert, and S. Limet, "ExaViz: a Flexible Framework to Analyse, Steer and Interact with Molecular Dynamics Simulations," *Faraday Discussions of the Chemical Society*, vol. 169, pp. 119–142, May 2014. [Online]. Available: https://hal.inria.fr/hal-00942627

[34] Morton, "A computer oriented geodetic data base and a new technique in file sequencing," Tech. Rep. Ottawa, Ontario, Canada, 1966.

[35] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, July 1968.

[36] M. Chavent, A. Vanel, A. Tek, B. Levy, S. Robert, B. Raffin, and M. Baaden, "GPU-accelerated atom and dynamic bond visualization using hyperballs: A unified algorithm for balls, sticks, and hyperboloids," *Journal of Computational Chemistry*, vol. 32, no. 13, pp. 2924–2935, 2011.

[37] T. Peterka, H. Croubois, N. Li, S. Rangel, and F. Cappello, "Self-Adaptive Density Estimation of Particle Data," *To appear in SIAM Journal on Scientific Computing SISC Special Edition on CSE'15: Software and Big Data*, 2016.

[38] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka *et al.*, "HACC: Simulating Sky Surveys on State-of-the-Art Supercomputing Architectures," *New Astronomy*, 2015.

[39] T. Peterka, D. Morozov, and C. Phillips, "High-Performance Computation of Distributed-Memory Parallel 3D Voronoi and Delaunay Tessellation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 997–1007.

[40] D. Morozov and T. Peterka, "Block-Parallel Data Analysis with DIY2," 2016.

[41] ——, "Efficient Delaunay Tessellation through K-D Tree Decomposition," in *Proceedings of SC16*. IEEE Press, 2016.